Submission Instructions Please submit a single text file (.py) containing Python code that contains the class MinHeap. If additional classes are necessary for your implementation, (e.g. HeapNode) those should be included as well. Please name the file as follows:

```
tcss501-hw03.py
```

Implement a MinHeap (20 pts)

A MinHeap is a common data structure used for efficient storing of ordered information in such a way that the lowest value in the data structure can be accessed in constant $O(1)$ time, removed in logarithmic $O(logn)$ time, and insertion of values can be done in logarithmic $O(logn)$ time. The Heap should support numeric data types (integers, floats) and string values. It needn't support mixed data types (e.g. some elements being strings and others being integers).

To meet the requirements of this assignment, you may implement any internal / helper methods as desired, but must include the following method signatures as an interface. While it is common to implement a Heap using a dynamic array or list, **implement your MinHeap using a Binary Tree nodes that contain pointers to their left and right children**, and optionally a pointer to a node's parent (this can make the *bubble up* algorithm easier to implement).

```
def pop(self):
""" Returns and removes the minimum value from the heap.  If the heap is empty, return None.
    Must be implemented with no worse than O(log n) time complexity. """

def peek(self):
""" Return but do not remove the minimum value of the heap.  If the heap is empty, return None.
    Must be implemented with no worse than O(1) time complexity. """

def insert(self, data):
""" Insert the provided data onto the heap.  Should support integers and strings at least.
    Must be implemented with no worse than O(log n) time complexity. """

def clear(self):
""" Removes all elements from the heap.  Must be implemented in no worse than O(1) time. """
```

The following explains how to approach the implementation of each of the 4 required methods.

.peek() Return the value from the Root Node of the Data Structure.

.pop() Store the value of the Root Node in a variable to return at the end. Replace Root Node's value with the value of the "last element", removing that element from the data structure by removing its parent's link to it. Then, push the new value in the root node down to the appropriate spot in the data structure as instructed in lecture. Decrement the size. Return the stored value from the first step.

.insert(data) Create a new node in the appropriate location in the tree (last node of last level). Bubble the element up as far as necessary in the heap. Increment the size.

.clear() Set the Root Node to None. Reset the size to zero.

More Tips: Remember that a Binary Heap is a complete tree, any new nodes are added from left to right on the last level of the tree. Use the size to determine how to traverse to the "last element".

Implement helper functions for the "bubble up" and "bubble down" algorithms. Call those functions inside of pop and insert at the appropriate times.

Don't wait. This assignment is designed to be more difficult than the first two assignments. Do not wait until the last minute to start it.